9. Reading Techniques

Forrest Shull

Too little attention has been paid to the development of effective procedures for the individual defect detection phase of an inspection. Often, reviewers are left to their own implicit procedures and experience for detecting defects, which can be effective for more experienced reviewers but provides little guidance for the less experienced and inhibits learning about effective approaches. "Reading techniques" are a procedural approach to defect detection that addresses these issues. By focusing reviewers on the perspective of a downstream user of the document being reviewed, reading techniques represent an improvement over ad hoc approaches. This chapter presents a description of reading techniques and guidelines for creating and tailoring them.

9.1 CURRENT PRACTICES

In the current state of the practice, software developers often rely on ad hoc techniques for their individual review of a software document, to read the document and detect defects. An ad hoc approach is one that is nonsystematic, nonspecific, and nondistinct.

Nonsystematic means that inspectors have no set procedure or rules to follow; they use whatever processes or tools seem useful to them at any given time. While this may seem like an appealing approach (it asks inspectors to rely on their own practical experience to decide how to inspect the document), there are some drawbacks. First, because there is no set procedure, it is difficult to provide training to inexperienced inspectors. New inspectors are required to build up their own experience over time; it is difficult for them to learn what other inspectors are already doing in order to build on their experience. Secondly, it is difficult to improve a process that is not even informally defined. If an inspector learns, for example, that a particular type of defect is consistently not being detected in the inspections, it is difficult for him or her to know how to alter the inspection process in order to more effectively detects defects of that type.

Nonspecific detection techniques mean that inspectors are required to find defects of all types. Again, at first this may seem like a benefit since all types of defects have to be removed from the document. But from the viewpoint of an individual inspector, a lack of a specific goal can make the defect detection process harder. When an inspector is equally responsible for finding typos as well as all kinds of omitted functionality, it becomes harder to do the search because the inspector is looking for many different things. Limiting the search makes the inspector responsible for only a subset of the defects, but allows that inspector to spend more energy and concentration looking for fewer types.

Nondistinct techniques mean that all inspectors have the same goals and responsibilities. In short, there is some duplication of work since all inspectors are reading the document looking for the same things. This approach does have its advantages; some defects are more likely to be found since there are more inspectors examining the same material looking for the same things. On the other hand, inspectors are motivated to do a better job when they knew that they are the only people responsible for a particular part of the inspection, and can't rely on other inspectors to find the defects they missed. Giving different inspectors distinct responsibilities also allows different inspectors to become "experts" in different aspects of a document, rather than requiring all inspectors to have the same, overview level of knowledge.

Inspections using rules and checklists represent an improvement over ad hoc approaches, largely by making the review process more specific and systematic. By providing focus for the inspectors, they can also support distinct roles or responsibilities. However, they still leave crucial information to be

discovered in an ad hoc way. For example, a checklist may say what issues to check for but not *how* an inspector gets the information to do so.

In contrast to other approaches, the use of a defined, systematic process for individual defect detection (known as a reading technique, since it helps reviewers to inspect, or read, a document more effectively) allows reviewers to better focus on the important aspects of the document being reviewed. More importantly, by making the review process explicit, reading techniques allow inspections to be adapted over time to better meet the needs of the organization. For example, if a particular type of defect is consistently missed by inspections, then a procedure for how that type of defect could be found should be developed and applied by at least one of the inspectors in future inspections. On the other hand, if a particular part of the procedure is felt by the inspectors to never lead to any defects being detected, then it might be dropped from the procedure, rather than continue to use inspectors' time for little gain.

This chapter provides an overview of reading techniques for software inspections. It provides a description of reading techniques and the underlying models, of the document being inspected and the types of defects sought, which are necessary for their use. A set of dimensions is presented along which any reading technique must be tailored to the specific organization doing the inspections, and examples are presented to illustrate the discussion. Finally, a summary of experiences with these techniques is discussed.

9.2 A READING PROCEDURE TEMPLATE

When reading techniques are used to perform an inspection, a family of related reading techniques is used in which each technique captures an important aspect of the information contained in the document under review. Each reading technique captures a different perspective on the information and all together the family of techniques should cover all the perspectives of interest. Since each inspector on the inspection team applies a particular reading technique, from a particular perspective, team members have unique (rather than redundant) responsibilities.

The use of perspectives reflects a belief that the correctness of software artifacts must be evaluated not according to some global and static definition of quality but according to how such documents will be used during the software lifecycle. That is, software artifacts are created not for any intrinsic value but in order to be used by some stakeholder at some later stage of software development. If these stakeholders can be identified and the artifact reviewed from their perspectives, then greater confidence can be had that the artifact will be able to meet all the downstream expectations for it. On the other hand, if any of these stakeholders' needs are not met, this represents a deficiency in the quality of the requirements that has the potential to negatively impact software development. This is what we refer to as a requirements defect.

Perspectives should be chosen so as to minimize the overlapping responsibilities among inspectors, while achieving a high level of coverage of the defects. This goal is illustrated by Figure 9.1, in which each of the tables represents the set of defects in some document, and an "X" in the column for that defect means it is targeted by one of the three review perspectives (Rev1, Rev2, or Rev3). The situation in which an ad hoc approach is used for individual defect detection is illustrated by the table on the left. In ad hoc reviews, each reviewer is asked to inspect the entire document but typically focuses on some subset of the possible defects, perhaps because the reviewer has more experience with a particular defect type, considers certain types of defects to be more important to find, or finds certain types of defects more easily. However, because this focusing is not made explicit, it often happens during the review that certain defects happen to be covered more thoroughly, by multiple reviewers (such as D2, D3, D4, D5), and other defects not at all (D7). The idea behind reading techniques is that each technique provides the reviewer with a process for reviewing the document from a certain perspective, and thus focuses the reviewer on a certain subset of defects. Perspectives

are chosen with the goal that all possible defects of interest are the target of at least one technique, with minimal overlap between them. This ideal situation is illustrated in the table on the right.

Ad hoc coverage:

	D1	D2	D3	D4	D5	D6	D7
Rev1	X	X	X				
Rev2		X	X	X	X		
Rev3				X	X	X	

Reading technique coverage:

	D1	D2	D3	D4	D5	D6	D7
Rev1	X	X					
Rev2			X	X			
Rev3					X	X	X

Figure 9.1: Simulated defect coverage of different inspection approaches, for three review perspectives (Rev1…Rev3) over a set of defects (D1…D7)

For example, for an inspection of requirements in a particular environment the following perspectives were found to be useful:

- The perspective of a designer, who uses the requirements as the basis for the system design;
- The perspective of a tester, who uses the requirements to determine whether the system exhibits proper behavior for each test case;
- The perspective of the customer, who has certain needs for the system functionality which are described in the requirements.

Each reviewer is assigned only one perspective from which to inspect the document (i.e. the reviewers have distinct responsibilities) and focus on the particular defects relevant to that perspective (i.e. have a specific focus). If the perspectives have been chosen wisely then the union of these inspectors covers all of the relevant defects in this document with a minimum of overlap. More information on identifying perspectives is given in paragraph 9.5.

Inspectors receive guidance for staying within their perspective by means of an operational procedure, a set of process steps or guidelines that keep them focused on just the information in the artifact that is important for their perspective (allowing a systematic inspection preparation). For each perspective, the operational procedure is created by first identifying an abstraction of the document, i.e. a way of modeling only the information in the document that is important for the perspective. Choosing the right abstraction requires answering the question of *what* information should be important to the inspector. Next, a model of *how* that information can be used to detect defects is necessary. This requires deciding which quality aspects are of interest in the document and understanding how the abstraction addresses those aspects.

These two models (the “what” and the “how”) must be used to create procedures for the reviewers to follow. As shown in Figure 9.2, the reading technique for reviewing the document is a combination of these specific procedures. The procedure that says what to look for is called the “abstraction procedure”; the procedure that says how to use the abstraction to look for defects is the “use procedure.”

An abstraction procedure is necessary to describe how the abstraction should be built. This procedure can be thought of as a “traversal” algorithm that instructs the reviewer in how to read through the document and find the information necessary for building the abstraction. Creating the abstraction procedure usually involves determining what are the relevant organizing structures for the given document from the reviewer’s perspective. This information helps the reviewer concentrate on just the relevant pieces of the document rather than the document as a whole. For example, if a requirements document is reviewed from the perspective of a tester, the reviewer needs to worry about how to make each requirement testable. The organizing structure is thus the requirements themselves, and the abstraction procedure could step through each requirement at a time, and help the reviewer to identify the relevant aspects, such as the expected inputs, the different classes of input values, and the expected outputs. For the perspective of a user, however, thinking about individual requirements

might not be very useful, so instead the abstraction procedure could help the reviewer to think about the larger functionalities involved and to abstract modes of behavior from the requirements. For a user, these modes of behavior can be easier to analyze for defects than lists of individual requirements.

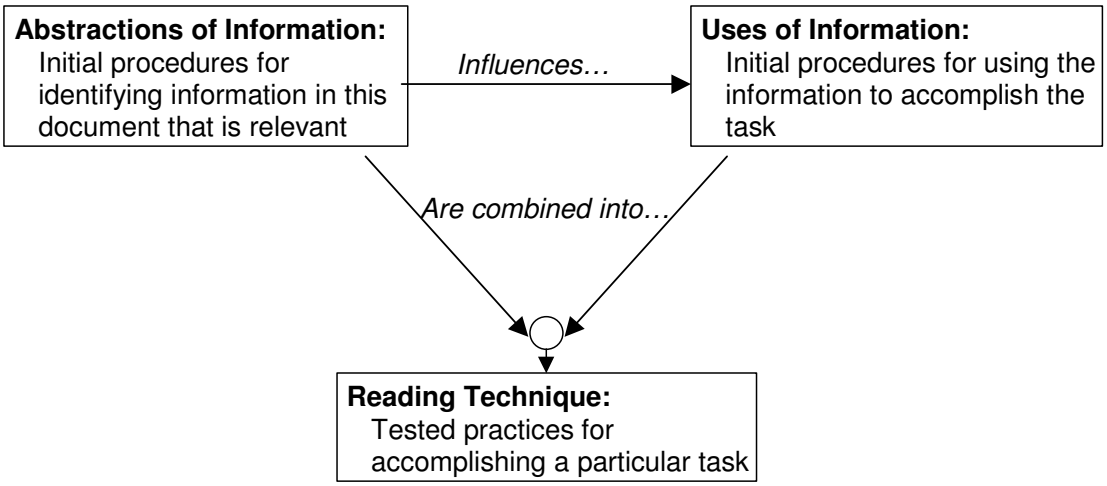

Figure 9.2: Building focused, tailored software reading techniques

Next, for each of the steps of the abstraction procedure generated above, a use procedure must be provided that contains guidance as to how the information seen so far can be checked for defects. The abstraction procedure gives the inspector the right information; the use procedure should provide a step-by-step way to check that information for defects. Once these procedures have been identified, the reading technique can be created by interleaving them in such a way that every step of the abstraction procedure is followed by a step of the use procedure, that uses the information just identified in the abstraction to look for defects.

9.3 EXAMPLE 1: THE REQUIREMENTS CUSTOMER PERSPECTIVE

This paragraph contains the full text from a reading technique created for the perspective of the customer in a requirements review. It represents the set of instructions given to a reviewer and corresponds to the outline for reading techniques described in paragraph 9.2, in which abstraction steps (here, the set of instructions that guide the reviewer through the process of building use cases) are followed by use steps (here, the sets of questions) that help the reviewer find defects in that information. It should be noted that this is only one possible instantiation for this perspective and that others could be created by tailoring along the dimensions identified in paragraph 9.5, for example, the level of detail, the abstraction chosen, or the defect taxonomy that yields the sets of questions. Another example of a reading technique is described in paragraph 9.4.

The goal for this procedure is to provide a comprehensive review of the quality aspects that would be relevant for the customer (i.e., to review whether the requirements correctly describe all of the necessary functionality for the system), regardless of quality aspects for other stakeholders of the requirements (e.g. whether the requirements are easily translatable into a design). This procedure includes a set of steps for developing use cases that describe the system functionality (the abstraction procedure), each of which is followed by associated questions (the use procedure). For example, Step 1 includes guidelines for identifying system "participants," a necessary first step toward developing the system abstraction, the use cases. The series of questions immediately following aim to uncover any defects in the information about the participants in the requirements.

The procedure:
"Create use cases to document the functionality that users of the system should be able to perform. This will require listing the functionality that the new system will provide and the external interfaces that are involved. You will have to identify actors (sets of users) who will use the system for specific types of functionality. Remember to include all of the functionality in the system, including special/contingency conditions. Follow the procedure below to generate the use cases, using the questions provided to identify faults in the requirements:

1) *(abstraction step)* **Read through the requirements once, finding participants involved.**
   a) Identify and list the participants in the new requirements. Participants are the other systems and the users that interact with the system described in the requirements – that is, they participate in the system functionality by sending messages to the system and/or receiving information and instructions from it. You may use the following questions to help you identify participants:
      - Which user groups use the system to perform a task?
      - Which user groups are needed by the system in order to perform its functions? These functions could be its major functions, or secondary functions such as system maintenance and administration.
      - Which are the external systems that use the system in order to perform a task?
      - Which are the external systems that are managed or otherwise used by the system in order to perform a task?
      - Which are the external systems or user groups that send information to the system?
      - Which are the external systems or user groups that receive information from the system?

   (use step)
   Q1.1 Are multiple terms used to describe the same participant in the requirements?
   Q1.2 Is the description of how the system interacts with a participant inconsistent with the description of the participant? Are the requirements unclear or inconsistent about this interaction?
   Q1.3 Have necessary participants been omitted? That is, does the system need to interact with another system, a piece of hardware, or a type of user that is not described?
   Q1.4 Is an external system or a class of "users" described in the requirements, which does not actually interact with the system?

2) *(abstraction step)* **Read through the requirements a second time, identifying the product functions.**
   a) Identify and record the set of functionality that the system must be able to perform. That is, what activities do the requirements say the system must do? (E.g. display a database record, print a report, create and display a graph.)
   b) Now consider how a user of the system will view it. He or she is probably not concerned with the individual activities that the system can perform, but instead thinks about using the system to achieve some goal (e.g. adding information to an existing database, computing a payment, viewing the current status of an account). These user goals will be the set of use cases for the system.
   c) For each use case, decide which of the system activities you previously recorded are involved, and note them. Sketch out, from the user's point of view, what steps are required to achieve the goal and use arrows to identify the flow of system control ("First this functionality happens, then this…"). Use branching arrows to signify places where the flow of control could split. Remember to include exception functionality in the use case. Check the appropriate functional requirements to ensure that you have the details of the processing and flow of control correct.
   d) For each use case you create, remember to signify what class(es) of users would use the functionality (the user classes should be already recorded in the list of participants), as well as the action that starts the functionality (e.g. "selecting option 2 from the main menu" might

start a use case for deleting records from a database). Record both of these pieces of information with the use case. Finally, give the use case a descriptive name that conveys the functionality it represents and assign it a number for future reference.

(use step)

Q2.1 Are the start conditions for each use case specified at an appropriate level of detail?

Q2.2 Are the class(es) of users who use the functionality described, and are these classes correct and appropriate?

Q2.3 Is there any system functionality that should be included in a use case but is described in insufficient detail or omitted from the requirements?

Q2.4 Has the system been described sufficiently so that you understand what activities are required for the user to achieve the goal of a use case? Does this combination of activities make sense, based on the general description and your domain knowledge? Does the description allow more than one interpretation as to how the system achieves this goal?

Q2.5 Do the requirements omit use cases that you feel are necessary, according to your domain knowledge or the general description?

3) *(abstraction step)* **Match the participants to all of the use cases in which they are involved.** (Remember that if two participants are involved in all of the same use cases, they might represent a single unique actor and should be combined.)

(use step)

Q3.1 Is it clear from the requirements which participants are involved in which use cases?

Q3.2 Based on the general requirements and your knowledge of the domain, has each participant been connected to all of the relevant use cases?

Q3.3 Are participants involved in use cases that are incompatible with the participant's description?

Q3.4 Have necessary participants been omitted (e.g., are there use cases which require information that cannot be obtained from any source described in the requirements)?"

9.4 EXAMPLE 2: THE REQUIREMENTS TEST PERSPECTIVE

As a further example, this paragraph provides an instantiation of a reading technique for requirements review representing the perspective of a tester. It represents the set of instructions given to a reviewer and corresponds to the outline for reading techniques described in paragraph 9.2. The goal of this perspective is to evaluate whether the functionality contained in the requirements is well-specified (i.e. that test plans can be constructed to demonstrate the adherence of the system to the requirements) regardless of quality aspects for other stakeholders (e.g. whether the complete set of functionality has been described).

As with the customer perspective given in 9.3, it should be noted that this example is only one possible instantiation for this perspective and that others could be created by tailoring along the dimensions identified in paragraph 9.5.

The procedure

For each requirement, generate a test or set of test cases that allow you to ensure that an implementation of the system satisfies the requirement. Follow the procedure below to generate the test cases, using the questions provided to identify faults in the requirements:

1) *(abstraction step)* **For each requirement, read it through once and record the number and page on the form provided, along with the inputs to the requirement**.

(use step)

Q1.1 Does the requirement make sense from what you know about the application or from what is specified in the general description?
Q1.2 Do you have all the information necessary to identify the inputs to the requirement? Based on the general requirements and your domain knowledge, are these inputs correct for this requirement?
Q1.3 Have any of the necessary inputs been omitted?
Q1.4 Are any inputs specified which are not needed for this requirement?
Q1.5 Is this requirement in the appropriate section of the document?

2) *(abstraction step)* **For each input, divide the input domain into sets of data (called equivalence classes), where all of the values in each set will cause the system to behave similarly.**
Determine the equivalence classes for a particular input by understanding the sets of conditions that affect the behavior of the requirement. You may find it helpful to keep the following guidelines in mind when creating equivalence classes:

- If an input condition specifies a range, at least one valid (the set of values in the range) and two invalid equivalence sets (the set of values less than the lowest extreme of the range, and the set of values greater than the largest extreme) are defined.
- If an input condition specifies a member of a set, then at least one valid (the set itself) and one invalid equivalence class (the complement of the valid set) are defined.
- If an input condition requires a specific value, then one valid (the class containing the value itself) and two invalid equivalence classes (the class of values less than, and the class greater than, the value) are defined.

Each equivalence class should be recorded with the appropriate input.

(use step)
Q2.1 Do you have enough information to construct the equivalence classes for each input? Can you specify the boundaries of the classes at an appropriate level of detail?
Q2.2 According to the information in the requirements, are the classes constructed so that no value appears in more than one set?
Q2.3 Do the requirements state that a particular value should appear in more than one equivalence class (that is, do they specify more than one type of response for the same value)?
Q2.4 Do the requirements specify the result for an invalid equivalence classes?

3) *(abstraction step)* **For each equivalence class write test cases, and record them beneath the associated equivalence set on the form.**
Select typical test cases as well as values at and near the edges of the sets. For example, if the requirement expects input values in the range 0 to 100, then the test cases selected might be: 0, 1, 56, 99, 100. Finally, for each equivalence class record the behavior, which is expected to result (that is, how do you expect the system to respond to the test cases you just made up?).

(use step)
Q3.1 Do you have enough information to create test cases for each equivalence class?
Q3.2 Are there other interpretations of this requirement that the implementor might make based upon the description given? Will this affect the tests you generate?
Q3.3 Is there another requirement for which you would generate a similar test case but would get a contradictory result?
Q3.4 Can you be sure that the tests generated will yield the correct values in the correct units? Is the resulting behavior specified appropriately?

9.5 CUSTOMIZING READING TECHNIQUES

Paragraph 9.2 provided a generic outline for how reading techniques are constructed. To be effective in practice, however, reading techniques must be tailored to the needs of the project and the organization. In the following subsections, several dimensions are identified along which such customization should be performed. In performing the customization, it is helpful to reuse prior knowledge whenever possible. That is, the reading technique should borrow from other methods, or incorporate tools, that have already been proven successful in the environment. In some cases, ethnographic studies may be necessary to understand exactly what techniques and tools developers have already found to be useful, for example to uncover particular types of defects that reviewers have determined are important, or a particular order for doing the analysis of the document that avoids repetition, or an existing tool that automates some portion of the checking. This attention to tested tools can result in a more effective technique for the environment, and aid in overcoming developer resistance at being asked to use a "new" technique.

Tailoring Perspectives

To detect defects in a document that would hinder its downstream use in software development, it is necessary to properly identify all of the stakeholders in later development phases. This selection of stakeholders has to be done according to organization or project needs. To do so, answering the following questions can be helpful:

- In what other phases of the software lifecycle is this document needed?
- In each phase, what other specific software tasks does the document support?
  For example, for a requirements inspection the following simple waterfall model of the software lifecycle could be sketched (figure 9.3).

Thinking about how the requirements document is used in each of these phases could yield the following:

- As a description of the needs of the customer: The requirements describe the set of functionality and performance constraints that must be met by the final system.
- As a basis for the design of the system: The system designer has to create a design that can achieve the functionality described by the requirements, within the allowed constraints.
- As a point of comparison for system test: The system's test plan has to ensure that the functionality and performance requirements have been correctly implemented.

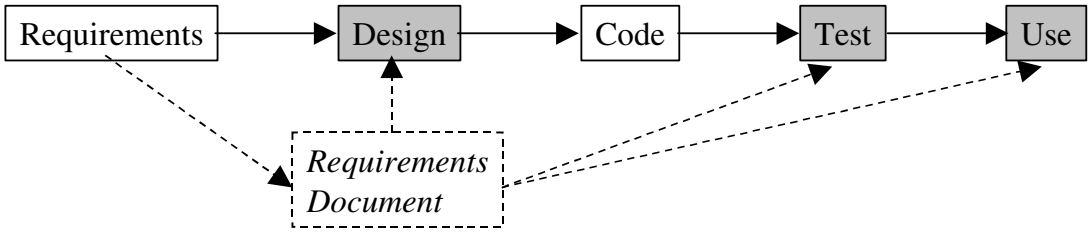

Figure 9.3: Simple model of requirements' use in the lifecycle

These uses suggest perspectives from which the requirements document should be reviewed. The designer needs a set of requirements that are correct, and described in sufficient detail for the major system components to be identified and correctly designed. The tester cares about whether the requirements are testable, and described in sufficient detail that a test plan, or set of test cases for the entire system, can be created. The customer (or user) of the system requires that the functionality he or she needs the system to have is completely and correctly captured by the requirements.

However, these are by no means the only perspectives that can be appropriate for a requirements review. The simple lifecycle model should be used as a tool to help identify the right perspectives for a given environment. For example, it may be that the needs of the stakeholders already identified (designer, tester, and customer perspectives) are not entirely applicable to another environment, and should be modified or dropped. Or, there might be a stakeholder identified in the coding phase (e.g. someone who has to integrate the functionality described for the new system with existing

components). It is also possible that other phases need to be added to the lifecycle model for a given environment. For example, if the system was being developed incrementally, such that multiple future versions of the system are planned, a maintainer perspective could be added that is responsible for reviewing the requirements for their extensibility (i.e. for evaluating how easy the next extension to such a system would be). The right mix of perspectives depends on the expected uses of the document being reviewed in the organizational environment.

Tailoring Abstraction Models

Using the right abstractions for a set of reading techniques is especially crucial because the abstractions created, when a reviewer applies the techniques during a review, will be used in two ways. First, they have to support defect detection as they are created. Therefore the abstractions chosen should be as simple as possible, so that the effort required for their construction does not distract from their use as a tool for finding defects. But, secondly, they must also be abstractions that are useful in some later lifecycle phase for the development of the system. That is, the effort expended to analyze the system and create the abstraction should also directly support system construction.

For example, in the previous paragraph designer, tester, and customer perspectives were discussed as examples for a requirements review. During the inspection, reviewers in each perspective could be asked to translate the information in the requirements into a high-level design sketch, a high-level test plan, and a set of use cases, respectively. Each of these abstractions was chosen because it gives the reviewer a new perspective on the information contained in the requirements relevant to that stakeholder (e.g. translating the requirements into use cases helps the "customer" reviewer think about how the functionalities specified will be implemented). The abstractions were also chosen for their usefulness in constructing the system. As shown in figure 9.4, use cases and design notes can be used in the design phase as a basis for system design, use cases can also be useful as the basis for a user manual, and the high-level test cases can be used as a starting point for the test phase.

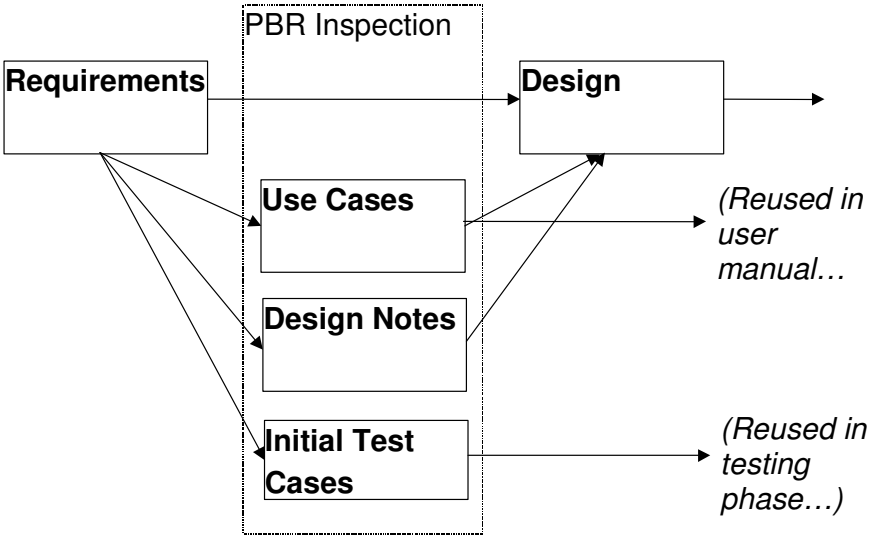

Figure 9.4: Use of abstractions created during a review elsewhere in the lifecycle.

Perhaps most importantly, abstractions should be chosen with which the likely reviewers have some expertise. Often, in organizations just introducing inspections, there are likely to be more personnel with expertise in creating the system abstractions (e.g. test plans) than in looking for requirements defects. Using the test case as an abstraction for a reading technique helps reviewers leverage their existing experience for a new goal.

The following questions will be helpful for identifying the right abstractions:

- What information must the document contain to support each perspective identified?

- What is a good way of organizing that information, so that attributes of interest are readily apparent to the inspector?
- What artifacts are being created in other lifecycle phases to support downstream development?

Tailoring the Defect Taxonomy

The questions that the inspector is asked about the abstraction correspond directly to the types of defects being sought. In order to target questions effectively, it is useful to create a taxonomy of the important issues. This requires an analysis of exactly what types of issues are of most concern to the project and organization. What constitutes a defect is largely situation-dependent. For example, if there are strong performance requirements on a system, then any description of the system that might lead to those performance requirements being unfulfilled contains a defect; however, for other systems with fewer performance constraints the same artifacts could be considered perfectly correct. Similarly, the types of defects in a textual requirements document could be very different from what would be used for a graphical design representation.

As a starting point, however, a generic defect taxonomy can be identified which can then be instantiated for specific circumstances. One such taxonomy that has proven useful is based on the idea of the software development lifecycle as a series of transformations of a system description to increasingly formal notations, and adding increasing levels of detail. For example, a set of natural-language requirements can be thought of as a loose description of a system that is transformed into high- and low-level designs, more formal descriptions of the same basic set of functionality. Eventually these designs are translated into code, which is more formal still, but describes the same set of functionality (hopefully) as was set forth in the original requirements.

So what can go wrong during such transformations? Figure 9.5 presents a simplified view of the problem, in which all of the relevant information has to be carried forward from the previous phase into a new form, and has to be specified in such a way that it can be further refined in the next phase. The ideal case is shown by arrow 1, in which a piece of information from the artifact created in the previous phase of development is correctly translated into its new form in the artifact in the current phase. There is, however, the possibility that necessary information is somehow left out of the new artifact (omitted information, arrow 2) or translated into the new artifact but in an incorrect form (incorrect fact, arrow 3). In the current phase artifact, there is always the possibility that extraneous information has been entered (arrow 4), which could lead to confusion in the further development of the system, or that information has been specified in such a way as to make the document inconsistent with itself (inconsistent information, arrow 5). A related possibility is that information has been specified ambiguously, leading to multiple interpretations in the next phase (ambiguous information, arrow 6), not all of which may be correct or appropriate[1].

[1] Of course, figure 9.5 is a simplified view. In reality, many of the implied 1-to-1 mappings do not hold. There may be multiple artifacts created in each stage of the lifecycle, and the information in a particular phase can influence many aspects of the artifact created in the next phase. For example, one requirement from a requirements specification can impact many components of the system design. When notational differences are taken into account (e.g. textual requirements are translated into a graphical design description) it becomes apparent why performing effective inspections can be such a challenging task.

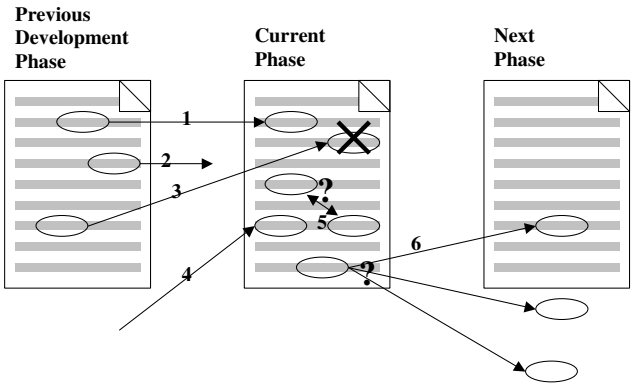

Figure 9.5: Representation of various defect types that can occur during software development

It is important to note that the classes are not orthogonal (i.e., a particular defect could possibly fit into more than one category) but are intended to give an idea of the set of possible defects that can occur. These broad categories need to be tailored to the document being inspected. For a requirements inspection, for example, the categories can be defined more exactly as:

- Omitted Information: some significant requirement related to functionality, performance, design constraints, attributes or external interface is not included, or the response of the software to all realizable classes of input data in all realizable classes of situations is not defined.
- Incorrect Fact: a requirement asserts a fact that cannot be true under the conditions specified for the system.
- Extraneous Information: information is provided that is not needed or used.
- Inconsistent Information: two or more requirements are in conflict with one another.
- Ambiguous Information: a requirement has multiple interpretations due to multiple terms for the same characteristic, or multiple meanings of a term in a particular context.

The important point is not the defect taxonomy itself, which is probably similar to the information given to reviewers during "traditional" inspections and which is expected to be tailored to the needs of a particular environment anyway. In a reading technique, the taxonomy is not given directly to the reviewer (except as supplemental material), but used instead to create a set of questions aimed at eliciting defects, for each step of the abstraction procedure. Taken as a whole, the reading technique provides the reviewer with a tested process to follow in order to answer the questions and find defects, rather than leaving it up to the reviewer how to look for defects of each type.

To illustrate, consider step 1 of the reading technique shown in paragraph 9.3. Following an abstraction step for eliciting information about participants comes a set of questions that check that information for ambiguous information (Q1.1), inconsistent information (Q1.2), omitted information (Q1.3), and extraneous information (Q1.4). The question list does not include defects of incorrect fact because that type of defect was not felt to be relevant for the information about participants, but other questions in this procedure (e.g. Q2.4) do search for that defect type in other information.

It is worth repeating that this is not a definitive taxonomy of defects, but a tool that can be used to start reasoning about the types of issues of relevance to a particular organization. Clearly, the taxonomy can be augmented (e.g. with performance issues) or shortened (e.g. if the organization does not find extraneous information to be important to consider in an inspection) or replaced altogether.

Tailoring the Level of Detail

The level of detail used in the procedure must also be seen as a dimension along which reading techniques can be customized to a particular organization, in order to yield higher effectiveness. Most importantly, the level of detail needs to vary with the expertise of the inspectors who will be applying the techniques: experienced reviewers should be free to make use of their own hard-won experience, while inexperienced reviewers may be more comfortable with more step-by-step guidelines.

Recall from paragraph 9.2 that the steps of a reading technique interleave abstraction steps (that help the reviewer identify information and build a model) and use steps (that give the reviewer a set of questions to answer about that information). Typically, varying the level of detail means varying the number of abstraction steps (and adjusting the associated use steps accordingly). For example, for experienced reviewers in the customer perspective there might be a single abstraction step ("Use whatever technique you normally do to reason about functionality") and a single associated use step ("Keep the following questions in mind while you do so…"). For less experienced reviewers a very specific abstraction may be chosen (e.g. use cases) so that multiple abstraction steps can be identified each with an associated set of questions.

Although more detailed procedures can be very useful for less experienced reviewers, there are tradeoffs to keep in mind. First, no procedure no matter how detailed will be effective if the reviewers don't have enough training to make effective use of the abstraction. Thus, the technique demonstrated in paragraph 9.3 will not be effective if the reviewers that use it have not received at least minimal training with use cases. Secondly, choosing a very detailed abstraction means reducing flexibility. Suppose, for the requirement review tester perspective, that "equivalence partitioning test cases" (BS7925-2, 1998) are selected as the abstraction. The resulting procedure may be useful for novice reviewers but some flexibility has been lost; not every requirement is well-suited to the equivalence partitioning approach. The lack of applicability of the more detailed procedure to some requirements has to be balanced against how well the inexperienced reviewers could do if allowed to select their own test approach on a requirement-by-requirement basis.

It is important – for both the software engineers tailoring the reading techniques and the inspectors applying them – to keep in mind the purpose of the model-building: the models are not important in themselves but as tools to find defects. It is sometimes tempting for software engineers to make the techniques too detailed, or for reviewers to spend too much time on model creation during a review, adding more detail than is necessary for finding defects in the document being inspected. Unfortunately there are no hard-and-fast guidelines that can be given; developers must use their own judgment as to when additional detail will not help find additional defects.

Tailoring Team Composition

A more advanced level of tailoring involves changing the composition of the inspection team. Typically, teams are composed of one member for each perspective, so that all important aspects of the document are covered by at least one member of the team. As data is collected concerning the effectiveness of a particular set of reading techniques for an organization, however, it becomes possible to better tailor team composition to larger organizational goals. For example, suppose the customer perspective for requirements review is found to be especially effective for finding defects in which important functionality has been omitted (reviewing from the customers' perspective helps focus on whether all the functionality they expect will be there). Then, organizations which have a history of problems with omitted functionality, or projects for which omitted functionality would be especially problematic, could add extra reviewers from this perspective to improve the chances that these important issues would be found.

Such tailoring cannot be done without enough measurement having been performed to be able to draw connections between the individual perspectives and their effectiveness for types of defects. For this reason, measurement and a well thought-out defect taxonomy are especially important for improving inspection practices.

9.6 EXPERIENCES

This chapter has concentrated on the family of reading techniques for requirements inspections (known as Perspective-Based Reading, or PBR). PBR focuses requirements inspection on catching inconsistent or incorrect requirements before they form the basis for design or implementation,

necessitating rework, and on verifying that information is complete and correct enough to support all stakeholders in downstream phases of development.

A number of studies have been run to assess the effectiveness of PBR in comparison to less procedural review techniques. Researchers in the U.S. (Shull, 1998), Norway (Sørumgård, 1997), and Germany (Ciolkowski *et al*, 1997) have run studies in university classes using over 150 software engineering students to evaluate and evolve the techniques. These students have run the gamut from undergraduates, with little previous review experience, to professionals with over 20 years of experience in industry who were returning for advanced degrees. In 1995 and 1996, studies were run using 25 professional developers from the NASA Goddard Space Flight Center (Basili et al, 1996). These developers were asked first to apply the requirements review technique they used at NASA, then were trained in PBR and asked to apply the new technique to a similar requirements document. In this way, researchers could assess how well these professionals performed using PBR compared with how they performed using their usual review technique.

These studies support the notion that PBR leads to improved rates of defect detection for both individual reviewers and review teams, for unfamiliar application domains. For example, in the 1995 NASA study, teams using PBR to review one document did on average 30% better than teams using their usual approach (finding 62% versus 48% of all defects in the document). For this document, individual reviewers also did significantly better when using PBR, with a 30% improvement over the non-PBR detection rate. On a second document, on which PBR and non-PBR individual reviewers performed about the same, PBR teams still had a higher effectiveness (finding 8% more of the total defects), perhaps because the use of perspectives reduced overlap between the reviewers. The NASA study also demonstrated that, for a familiar application domain, one danger is that experienced reviewers ignore the PBR procedure and go back to using the heuristics they had previously acquired. For this type of reviewer, training and reinforcement are needed to overcome this tendency.

Another benefit from these studies is that, by observing the use of PBR in a large number of environments and by large numbers of reviewers, researchers better understand the effects of PBR in different contexts and for different types of users. For example, PBR seems best suited to reviewers with a certain range of experience. Reviewers who have previously inspected requirements documents on multiple industrial projects have, over time, typically developed their own approaches, and do not benefit significantly from the introduction of PBR. Reviewers with very low experience (i.e. who have never been trained, or have been trained but never applied on a real project) with the relevant work products (e.g. designs or test plans) need to receive sufficient training before they can effectively apply PBR (Shull, 1998). This training seems to be necessary so that the difficulties of creating the representation of the system do not distract from the process of checking for defects.

Reading techniques are procedural guidelines that can be used by reviewers to improve the effectiveness of the individual defect detection phase of software inspections. Typically, reading techniques ask each reviewer of a document to take on the viewpoint of a downstream user of that document, looking for issues that would hinder its use from their perspective. To help keep reviewers focused on their perspective, reading techniques guide the reviewer through a relevant abstraction of the information in the document (e.g. through a set of use cases, for a customer perspective) and provides questions aimed at eliciting defects in that information.

The reading technique approach has been used to develop and refine multiple sets of reading techniques. This text has concentrated on the family of reading techniques for requirements inspections (known as Perspective-Based Reading, or PBR) but the idea has been tailored to other document types as well (Shull, 2002), most recently for high-level Object-Oriented designs (using Object-Oriented Reading Techniques or OORTs).

REFERENCES

Basili, V., S. Green, S., O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz (1996), The Empirical Investigation of Perspective-Based Reading, in*: Empirical Software Engineering*, 1996, pp. 133-164.

BS 7925-2 (1998), *Software Component Testing*, British Standards Institution

Ciolkowski, C., C. Differding, O. Laitenberger, and J. Muench (1997*), Empirical Investigation of Perspective-based Reading: A Replicated Experiment*, Technical Report ISERN-97-13, International Software Engineering Research Network.

Shull, F. (1998*), Developing Techniques for Using Software Documents: A Series of Empirical Studies*, Diss. Computer Science Department, University of Maryland

Shull, F. (2002), Software Reading Techniques*,* in *Encyclopedia of Software Engineering*, ed. John J. Marciniak, John Wiley & Sons.

Sørumgård, S. (1997*), Verification of Process Conformance in Empirical Studies of Software Development*, Diss. Norwegian University of Science and Technology